

每周工作汇报

姓名	侯宇轩	日期	2019.7.1-2019.7.8
----	-----	----	-------------------

1. 本周工作

0. 时间线（标黄的为现在正在处理，标绿为已完成）

第一阶段：测试数据+CPU

1.1 准备一个浮点数测试数据。（目前小鼠数据是灰度值数据，可以直接将 90M 测试集转为浮点数先用）（发现可能不需要浮点数，直接使用整数（灰度值）的数据转为 YUV 格式）

1.2. 将其在 CPU 上转换为 YUV 4: 2: 0 格式。（目前的方法：直接修改扩展名）
转换时通过补齐片数到 3 的倍数修正了 YUV420 格式片数不是整数片的问题

1.3. 将得到的 YUV 格式数据在 CPU 上转换为 H.264 格式。（使用他人的代码，已找到）

1.4 将得到的 H.264 格式解码为 YUV 格式。（使用他人的代码，已找到）
decode 时出现的丢帧情况已解决

1.5 将 YUV 格式转回为浮点数测试数据。（目前的方法：直接修改扩展名）

1.6 进行比较：使用压缩解压后数据/未压缩数据造成的渲染质量的不同。

其中，1.1-1.5 需要约一周时间（6.14-6.21），1.6 需要 2 天时间（6.22,6.24）

第二阶段：测试数据+GPU

2.1 寻找 1.4 步骤的代替，设法在 GPU 上将 H.264 格式转回为 YUV 格式(Decoder)。

目前，根据任老师推荐的 Nvidia Decoder Codec SDK, 细分任务如下：

2.1.1 安装该 SDK, 运行其例子，了解其工作流程

2.1.2 编写调用 Decoder 的代码

2.1.3 使用小的真数据进行测试 Decoder（若不行，可以再换假数据测试）

2.1.4 使用更大的数据测试 Decoder，最大：4096*4096*4096

2.2 进行同 1.6 的测试。

2.3 将浮点数->YUV->H.264->YUV->浮点数的流程组成完整的代码。

与陶老师交流后，提出了新的要求：

目前，对 Encoder/Decoder 的调用是从文件到文件的（即：给定一个编码后的文件，输入 Decoder，输出一个解码后的文件），实际上这样的操作并不能加快体绘制的速度，因为体绘制原本要从硬盘中读取文件，现在还是要从硬盘中读取文件。

因此，陶老师希望得到的效果是，编码（压缩）后的文件通过 Decoder 后直接留在 GPU 中，用于直接从 GPU 中进行体绘制。

与做小鼠脑影像体绘制的师兄交流后，得知他的设想如下：

方案 1：

1）将压缩后的体数据分块排布（在内存中）。

接口：可以通过分块的编号得到其对应压缩后体数据的位置与大小。

2）使用 OPENGL 将其读入并加入 GPU。

3）在 GPU 中解压缩。

4）利用 OPENGL 绘制。

方案 2：

1）将解压缩数据分块排布在 GPU 中。（分块模式与方案 1 的 1）相同）

2）利用 OPENGL 绘制。

方案 2 的 1）相当于将方案 1 中的（1,2,3）合并。

无论哪一种，由于解压缩是使用 CUDA 的，而绘制是使用 OPENGL 的，这两种库要想结合使用都需要一定的工程量。但无论哪种，都需要以下工作：

2.3.1 将数据分块（或者：压缩后进行分块排列）。

2.3.2 将封装度很高的 NVIDIA CODEC SDK 中的解码器从 文件->文件 提取出块数据->块数据的功能。

2.4 寻找 1.3 步骤的代替，设法在 GPU 上将 YUV 格式转为 H.264 格式(Encoder)。

其中，~~2.1 需要两周时间，(6.24-7.8)~~

~~2.2、2.3 需要一周时间 (7.9-7.15)。~~

~~2.4 需要一周时间。(7.16-7.22)~~

2.3 需要两周时间。 (7.8-7.22)

第三阶段：大数据+GPU




3.1 设计如何分块处理大数据.....

2. 本周工作：

完成了 2.2、2.4，测试了 GPU Encoder，测试了 Decode 后与原图的区别。

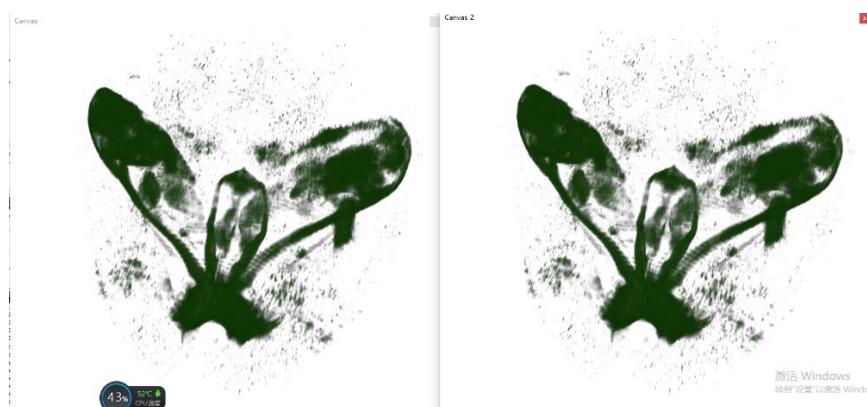
对于 GPU Encoder，测试发现与之前 CPU Encoder 的结果相同。

对之前选定的质量-大小折中量化参数 $q=30$ 下 编码的数据进行解码：

 mouse890x684x152_lod5_added_q0_30.h264	2019/6/21 16:46	H264 文件	860 KB
 mouse890x684x152_lod5_added_q0_30_decoded	2019/6/21 16:49	RAW 文件	92,741 KB
 mouse890x684x152_lod5_added_q0_30_decoded_gpu	2019/6/29 16:58	RAW 文件	92,741 KB

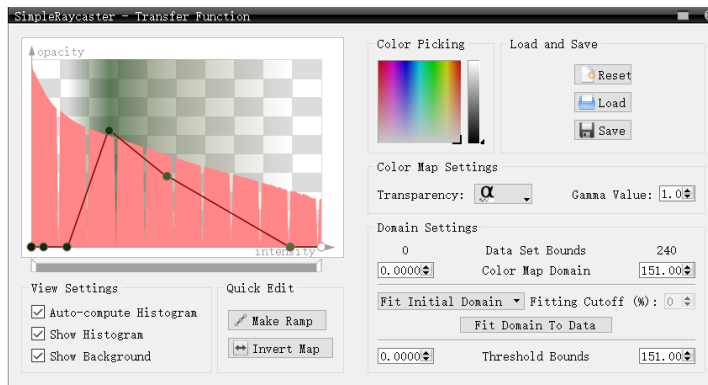
此时 H264 文件大小为 0.86M，比原来的 90M 压缩近 100 倍。

质量比较如下：



左图： CPU 解码结果 右图： GPU 解码结果

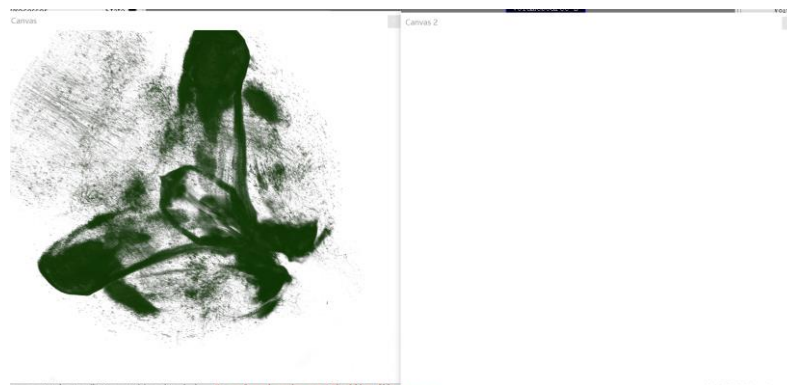
我们画上面的图，使用的 transfer function 如下：



图中左侧红色的柱形图是原图的直方图分布。折线的高度表示对对应灰度值的映射颜色深度，越高代表对应颜色越深，折线在水平线上则表示不画出对应区域。

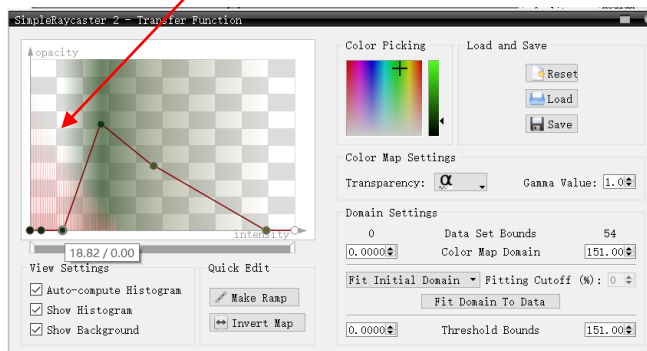
整个的灰度取值范围为 0(最左)-255(最右)。

仍然使用这个 transfer function，将原始数据与 q=30 解码后数据的差比较如下：



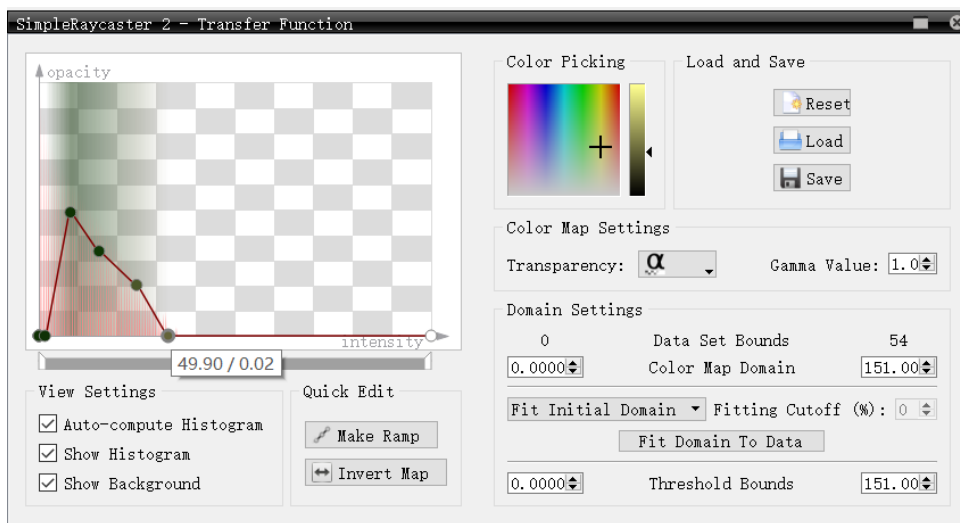
左：原图 右：差图（看起来是空白）

查看差图的直方图：（左上棋盘格中的细红线部分）



可以看到差的直方图与 transfer function 的映射相差较多，因此看起来是空白。

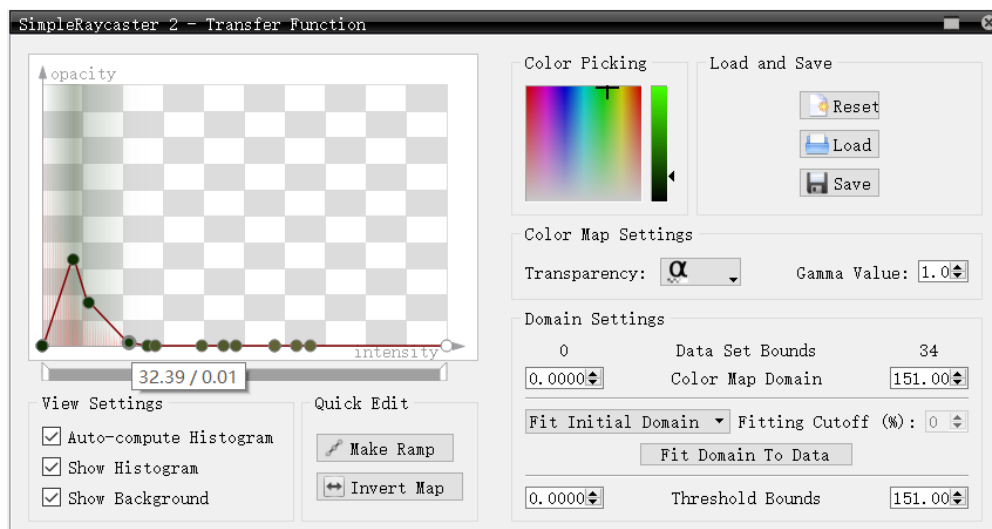
查看差图的直方图具体分布：



由点可知，差的最大值约为 50，主要分布在 20 以下（注：原图灰度为 0-255）。

不过，将原图（空白图的左侧）与 $q=30$ 解码图（本文的第一张图）对比，可以看到还是有不少的细节损失。

对于更精细的 $q=20$ （刚刚是 $q=30$ ， q 越小精度越高，压缩后文件越大），与原图作差后的直方图如下：



由图可知差的最大值约为 30，主要分布在 10 以下，比 $q=30$ 更加精细。

综上所述我们可以认为 $q=20$ 是一个比较能够接受的量化值。

2. 明日计划

继续 2.3，解读 GPU 解码的接口，试图从文件→文件转为块数据→块数据